Thoughts on Modelling Text in Nonpareil

Jeffrey H. Kingston

23 October, 2004

Abstract

Nonpareil is a document editing system in the early stages of development. This report contains some musings about modelling text in Nonpareil. Comments, including references to other work covering similar ground, are solicited.

1. Introduction

Nonpareil needs a file format for large literal objects, the equivalent of XML. But when I set out to define that format, I found that I needed an object-oriented model of text, and then I began to worry about whether that model was amenable to interactive editing. Hence this paper.

Nonpareil documents are more complex than traditional text in two ways. First, their content has a tree structure typified by equations:

$$\frac{x^2 + y^2}{2}$$

It is easy enough to select subtrees, but clicking at one point and dragging to an arbitrary second point is not always well-defined. For example, clicking half-way through the title box of a bibliographic reference entry form and dragging to halfway through the author box can't really be given a meaning, although windowing systems sometimes try to, by throwing away all structure and returning the concatenation of all the characters in the selection.

The other source of complexity is type. One should be able to select a chapter, or a sequence of chapters, for deletion, rearrangment, etc. Yet it would be wrong to insert an equation into a sequence of chapters. This would be a type error and it shows that the interface has to allow only operations that produce type-correct results.

2. Selection

Text editors are commendably simple in having just one type of selection. For any sequence, define the *selection points* of the sequence to be before the first element, between any two elements, and after the last element. An empty sequence has one selection point. A text editor manages a single sequence of characters, and a selection is a pair of selection points in that sequence, which may be equal, with the first not preceding the second.

Of course, there may be many ways to specify such a selection. Vi (nowadays, gvim) is my favourite editor because of all the ways you can specify selections (and because it unifies selection with moving the cursor). But in the end, what you get is a pair of selection points.

Selection in strongly typed tree structures is not so simple. Selection of a subtree is one possibility. A tree node may contain a sequence of children, that is, a variable of type *list*{x} for any x, and then sequence selection may be used on that sequence, with its elements treated atomically. Call this set of two types of selection *basic structure selection.* The question is, is basic structure selection enough?

3. Modelling text

Equation editing and large-scale structure editing are relatively simple because expectations of what can be selected correspond with basic structure selection, and expectations of what is a type-safe operation correspond with actual types – all subtrees of an equation have the same type, all large-scale structure symbols have the same type (we don't have separate *chapter*, *section* etc. types; these objects all have the same type, so moving a section into a list of chapters 'converts' it into a chapter, etc.).

Editing text is more difficult because user expectations are that text is an unstructured sequence of characters and embedded objects. If we impose more structure and more typing, then the user is in for some surprises.

One approach would be to mimic plain text and say that text is a sequence of characters and embedded objects that are sufficiently simple to stand alongside characters (not chapters or sections, for example). Some of the characters or objects denote paragraph breaks, sentence breaks, and line breaks. The main advantage of this approach is that it allows selection of any subsequence, including selections say of the last half of one word and the first half of the next, and since all these selections have the same type (sequence of character or simple object), any selection can be deleted or inserted at any selection point.

Although the editing operations offered immediately by this model are very desirable (and we would hope to offer them in Nonpareil), as a document model we reject this approach for two reasons. One is that it would lead to severe efficiency problems, given Nonpareil's functional approach, to hold (potentially) an entire document in a single list. When the list is edited at any point, all the earlier list nodes must be copied. This is bearable for a paragraph, say, but not for a whole document. The other reason for rejecting it is just the weakness of the model it provides. The system would be perpetually analysing the sequence to find word boundaries and so forth, essentially trying to discover the structure that is not officially there at all.

We have already seen that the model must be strongly typed – there cannot be just one type common to all displayable objects. The type structure should be as loose as possible; we don't want intuitively allowable operations to be prohibited by type barriers. But not too loose. Imagine inserting part of a word, considered as a sequence of displayables, between two paragraphs, considered as a sequence of two displayables. The logical result would make each fragment of the word into a paragraph – sure to frighten any user badly.

There probably is a general type containing all displayable objects, if only a weak one:

```
class displayable

noncreation

    show(e: env): display

end
```

But we want to model one particular type, *text*, specifically as a sequence of paragraphs:

```
class text inherit displayable

    paragraphs: list{paragraph}

end
```

A paragraph might be a list of sentences, or it might just be a list of words:

```
class paragraph inherit displayable

    words: list{word}

end
```

where a word is more general than just a sequence of characters; it is intended to be any object suitable for laying out on lines, and potentially hyphenatable:

```
class word inherit displayable

noncreation

    hyphenations(e: env): list{pair{word, word}}

end
```

Some words might be the usual sequences of atomic objects, intended to be displayed immediately adjacent to each other:

```
class atom_seq inherit word

    atoms: list{atom}

end
```

where an *atom* could be a sequence of characters, or some more general object encapsulated so that it behaves atomically (e.g. an unbreakable inline equation). Other words might be sentence separators, if we choose not to have sentences, or displays, which are considered to be part of the enclosing paragraph rather than in a paragraph of their own, since that is the usual logical structure.

It is fairly clear that a user interface can be built automatically for objects of the above classes, given implementations of *show* and *hyphenations.* Whenever lists are displayed it would offer selection points before and after each element; one could select either a subtree by clicking or a subsequence by click and drag. There would be a select point at the end of a word for appending to the word, another select point just to its right for inserting a new word, and a third one just to the right of that for inserting at the front of the next word.

But what would happen if, say, we selected a subsequence of the *atoms* object in *atom_seq* and attempted to insert it between two words? It should be doable, but the selection has type *list*{*atom*} while the insertion point has type *list*{*word*}. We need to coerce *list*{*atom*} first to *atom_seq*, then up to *word*, then coerce again to *list*{*word*}. We may also need to coerce downwards, for example to insert an entire word into the middle of another word.

These coercions are by no means impossible. The problem is that the arbitrary selections allowed by plain text are not allowed here. One can select two words but not the second half of the first word and the first half of the next, and so on. For such selections to be possible, we would need to go beyond basic structure selection, and operations would need to be able to handle the resulting more complex selections.

4. In-paragraph format changes

Format changes such as italics, underlining, or enclosure in a box raise the question of what type of object they apply to, and what type of result object they produce. An italicized word is still a word, an italicized paragraph is still a paragraph, suggesting that the type is

class *format*{*x* **is** *displayable*} **inherit** *x*

but Nonpareil's type system (like every other object-oriented type system) does not allow this, and for good reason. Yet separate *format_paragraph*, *format_word* etc. classes feel wrong and were not needed in Lout.

Thus, format changes push the design towards a single displayable object type. But should it be possible to italicize a whole sequence of paragraphs, for example? If so, who knows that the enclosed sequence is a sequence of paragraphs, requiring to be laid out in the paragraph manner, as part of the enclosing sequence of paragraphs?

Format changes within paragraphs may interfere with line breaking. Most systems (including Lout) allow font changes and underlining without such interference, but other changes, such as enclosure in a box, tend to produce an atomic result that cannot be broken or hyphenated as part of the enclosing paragraph, when in fact the user should have the choice.

The problem is not really one of implementation. A format change object can be queried for its break points and hyphenation points, and these can be tried by the enclosing paragraph. The problem is in the data model, in defining the type of objects to which format changes can be applied.

5. Multi-selection

This little document seems to contain more questions than answers, but to finish up here is something that I am pretty sure about.

Current editors known to me all have an exasperating restriction: they allow only a single selection at a time. Even basic cut-and-paste involves two selections (one specifying what to cut, the other where to paste). Excel sort of knows this. Because only one selection at a time is allowed, the user has to invoke two operations, the first to cut into the clip buffer, and the second to paste from there to a nominated spot. In X-windows you can combine the paste with the target selection using the middle mouse button, but you don't get to see your selection fixed before the paste goes in, and you can only copy, not move.

Whatever forms of selection are allowed in the end, Nonpareil will allow the user to make a sequence of selections. To do this, you would make one selection, then click on an icon to remember that selection, then make another, and so forth. (Just making a second selection would erase the first selection as usual, since mostly that is what's wanted.) Then buttons would be offered to move the first selection to the second, the second to the third, etc.; to rotate the selections around; and to permute them so that the selected order becomes the order in the document.

These operations would work on selections of any type. For example, you could bring up a representation of a sequence of chapters as a sequence of icons, select some permutation of some of them, and sort.

There would still be a clip buffer, but it would just be another document that you could copy into if you want to, not a special place.